Optimizing affinity-based binary hashing using auxiliary coordinates

Ramin Raziperchikolaei Miguel Á. Carreira-Perpiñán
Electrical Engineering and Computer Science, University of California, Merced
`http://eecs.ucmerced.edu`

February 4, 2016

Abstract

In supervised binary hashing, one wants to learn a function that maps a high-dimensional feature vector to a vector of binary codes, for application to fast image retrieval. This typically results in a difficult optimization problem, nonconvex and nonsmooth, because of the discrete variables involved. Much work has simply relaxed the problem during training, solving a continuous optimization, and truncating the codes a posteriori. This gives reasonable results but is quite suboptimal. Recent work has tried to optimize the objective directly over the binary codes and achieved better results, but the hash function was still learned a posteriori, which remains suboptimal. We propose a general framework for learning hash functions using affinity-based loss functions that uses auxiliary coordinates. This closes the loop and optimizes jointly over the hash functions and the binary codes so that they gradually match each other. The resulting algorithm can be seen as a corrected, iterated version of the procedure of optimizing first over the codes and then learning the hash function. Compared to this, our optimization is guaranteed to obtain better hash functions while being not much slower, as demonstrated experimentally in various supervised datasets. In addition, our framework facilitates the design of optimization algorithms for arbitrary types of loss and hash functions.

1 Introduction

Information retrieval arises in several applications, most obviously web search. For example, in image retrieval, a user is interested in finding similar images to a query image. Computationally, this essentially involves defining a high-dimensional feature space where each relevant image is represented by a vector, and then finding the closest points (nearest neighbors) to the vector for the query image, according to a suitable distance (Shakhnarovich et al., 2006). For example, one can use features such as SIFT (Lowe, 2004) or GIST (Oliva and Torralba, 2001) and the Euclidean distance for this purpose. Finding nearest neighbors in a dataset of N images (where N can be millions), each a vector of dimension D (typically in the hundreds) is slow, since exact algorithms run essentially in time $\mathcal{O}(ND)$ and space $\mathcal{O}(ND)$ (to store the image dataset). In practice, this is approximated, and a successful way to do this is *binary hashing* (Grauman and Fergus, 2013). Here, given a high-dimensional vector $\mathbf{x} \in \mathbb{R}^D$, the hash function $\mathbf{h}$ maps it to a b-bit vector $\mathbf{z} = \mathbf{h}(\mathbf{x}) \in \{-1, +1\}^b$, and the nearest neighbor search is then done in the binary space. This now costs $\mathcal{O}(Nb)$ time and space, which is orders of magnitude faster because typically $b < D$ and, crucially, (1) operations with binary vectors (such as computing Hamming distances) are very fast because of hardware support, and (2) the entire dataset can fit in (fast) memory rather than slow memory or disk.

The disadvantage is that the results are inexact, since the neighbors in the binary space will not be identical to the neighbors in the original space. However, the approximation error can be controlled by using sufficiently many bits and by *learning a good hash function*. This has been the topic of much work in recent years. The general approach consists of defining a supervised objective that has a small value for good hash functions and minimizing it. Ideally, such an objective function should be minimal when the neighbors of any given image are the same in both original and binary spaces. Practically in information retrieval, this is often evaluated using precision and recall. However, this ideal objective cannot be easily optimized over hash functions, and one uses approximate objectives instead. Many such objectives have been proposed in the literature. We focus here on *affinity-based loss functions*, which directly try to preserve the original similarities in the binary space. Specifically, we consider objective functions of the form

$$\min \mathcal{L}(\mathbf{h}) = \sum_{n,m=1}^{N} L(\mathbf{h}(\mathbf{x}_n), \mathbf{h}(\mathbf{x}_m);\ y_{nm}) \tag{1}$$

where $\mathbf{X} = (\mathbf{x}_1, \dots, \mathbf{x}_N)$ is the high-dimensional dataset of feature vectors, $\min_{\mathbf{h}}$ means minimizing over the parameters of the hash function $\mathbf{h}$ (e.g. over the weights of a linear SVM), and $L(\cdot)$ is a loss function that compares the codes for two images (often through their Hamming distance $\|\mathbf{h}(\mathbf{x}_n) - \mathbf{h}(\mathbf{x}_m)\|$) with the ground-truth value y_{nm} that measures the affinity in the original space between the two images $\mathbf{x}_n$ and $\mathbf{x}_m$ (distance, similarity or other measure of neighborhood; Grauman and Fergus, 2013). The sum is often restricted to a subset of image pairs (n, m) (for example, within the k nearest neighbors of each other in the original space), to keep the runtime low. Examples of these objective functions (described below) include models developed for dimension reduction, be they spectral such as Laplacian Eigenmaps (Belkin and Niyogi, 2003) and Locally Linear Embedding (Roweis and Saul, 2000), or nonlinear such as the Elastic Embedding (Carreira-Perpiñán, 2010) or t-SNE (van der Maaten and Hinton, 2008); as well as objective functions designed specifically for binary hashing, such as Supervised Hashing with Kernels (KSH) (Liu et al., 2012), Binary Reconstructive Embeddings (BRE) (Kulis and Darrell, 2009) or Semi-supervised sequential Projection Learning Hashing (SPLH) (Wang et al., 2012).

If the hash function $\mathbf{h}$ was a continuous function of its input $\mathbf{x}$ and its parameters, one could simply apply the chain rule to compute derivatives over the parameters of $\mathbf{h}$ of the objective function (1) and then apply a nonlinear optimization method such as gradient descent. This would be guaranteed to converge to an optimum under mild conditions (for example, Wolfe conditions on the line search), which would be global if the objective is convex and local otherwise (Nocedal and Wright, 2006). Hence, optimally learning the function $\mathbf{h}$ would be in principle doable (up to local optima), although it would still be slow because the objective can be quite nonlinear and involve many terms.

In binary hashing, the optimization is much more difficult, because in addition to the previous issues, the hash function must output binary values, hence the problem is not just generally nonconvex, but also nonsmooth. In view of this, much work has sidestepped the issue and settled on a simple but suboptimal solution. First, one defines the objective function (1) directly on the b-dimensional codes of each image (rather than on the hash function parameters) and optimizes it assuming continuous codes (in $\mathbb{R}^b$). Then, one binarizes the codes for each image. Finally, one learns a hash function given the codes. Optimizing the affinity-based loss function (1) can be done using spectral methods or nonlinear optimization as described above. Binarizing the codes has been done in different ways, from simply rounding them to $\{-1, +1\}$ using zero as threshold (Weiss et al., 2009; Zhang et al., 2010; Liu et al., 2011, 2012), to optimally finding a threshold (Liu et al., 2011; Strecha et al., 2012), to rotating the continuous codes so that thresholding introduces less error (Yu and Shi, 2003; Gong et al., 2013). Finally, learning the hash function for each of the b output bits can be considered as a binary classification problem, where the resulting classifiers collectively give the desired hash function, and can be solved using various machine learning techniques. Several works (e.g. Zhang et al., 2010; Lin et al., 2013, 2014) have used this approach, which does produce reasonable hash functions (in terms of retrieval measures such as precision and recall).

In order to do better, one needs to take into account during the optimization (rather than after the optimization) the fact that the codes are constrained to be binary. This implies attempting directly the discrete optimization of the affinity-based loss function over binary codes. This is a daunting task, since this is usually an NP-complete problem with Nb binary variables altogether, and practical applications could make this number as large as millions or beyond. Recent works have applied alternating optimization (with various refinements) to this, where one optimizes over a usually small subset of binary variables given fixed values for the remaining ones (Lin et al., 2013, 2014), and this did result in very competitive precision/recall compared with the state-of-the-art. This is still slow and future work will likely improve it, but as of now it provides an option to learn better binary codes.

Of the three-step suboptimal approach mentioned (learn continuous codes, binarize them, learn hash function), these works manage to join the first two steps and hence learn binary codes. Then, one learns the hash function given these binary codes. Can we do better? Indeed, in this paper *we show that all elements of the problem (binary codes and hash function) can be incorporated in a single algorithm that optimizes* *jointly over them.* Hence, by initializing it from binary codes from the previous approach, this algorithm is guaranteed to achieve a lower error and learn better hash functions. In fact, our framework can be seen as an iterated, corrected version of the two-step approach: learn binary codes *given the current hash function*, learn hash functions given codes, iterate (note the emphasis). The key to achieve this in a principled way is to use a recently proposed *method of auxiliary coordinates (MAC)* for optimizing "nested" systems, i.e., consisting of the composition of two or more functions or processing stages. MAC introduces new variables and constraints that cause decoupling between the stages, resulting in the mentioned alternation between learning the hash function and learning the binary codes. Section 2 reviews affinity-based loss functions, section 3 describes our MAC-based proposed framework, section 4 evaluates it in several supervised datasets, using linear and nonlinear hash functions, and section 5 discusses implications of this work.

Related work Although one can construct hash functions without training data (Andoni and Indyk, 2008; Kulis and Grauman, 2012), we focus on methods that learn the hash function given a training set, since they perform better, and our emphasis is in optimization. The learning can be unsupervised, which attempts to preserve distances in the original space, or supervised, which in addition attempts to preserve label similarity. Many objective functions have been proposed to achieve this and we focus on affinity-based ones. These create an affinity matrix for a subset of training points based on their distances (unsupervised) or labels (supervised) and combine it with a loss function (Liu et al., 2012; Kulis and Darrell, 2009; Norouzi and Fleet, 2011; Lin et al., 2013, 2014). Some methods optimize this directly over the hash function. For example, Binary Reconstructive Embeddings (Kulis and Darrell, 2009) use alternating optimization over the weights of the hash functions. Supervised Hashing with Kernels (Liu et al., 2012) learns hash functions sequentially by considering the difference between the inner product of the codes and the corresponding element of the affinity matrix. Although many approaches exist, a common theme is to apply a greedy approach where one first finds codes using an affinity-based loss function, and then fits the hash functions to them (usually by training a classifier). The codes can be found by relaxing the problem and binarizing its solution (Weiss et al., 2009; Zhang et al., 2010; Liu et al., 2011), or by approximately solving for the binary codes using some form of alternating optimization (possibly combined with GraphCut), as in two-step hashing (Lin et al., 2013, 2014; Ge et al., 2014), or by using relaxation in other ways (Liu et al., 2012; Norouzi and Fleet, 2011).

2 Nonlinear embedding and affinity-based loss functions for binary hashing

The dimensionality reduction literature has developed a number of objective functions of the form (1) (often called "embeddings") where the low-dimensional projection $\mathbf{z}_n \in \mathbb{R}^b$ of each high-dimensional data point $\mathbf{x}_n \in \mathbb{R}^D$ is a free, real-valued parameter. The neighborhood information is encoded in the y_{nm} values (using labels in supervised problems, or distance-based affinities in unsupervised problems). A representative example is the elastic embedding (Carreira-Perpiñán, 2010), where $L(\mathbf{z}_n, \mathbf{z}_m;\ y_{nm})$ has the form:

$$y^+_{nm} \left\|\mathbf{z}_n - \mathbf{z}_m\right\|^2 + \lambda y^-_{nm} \exp\left(- \left\|\mathbf{z}_n - \mathbf{z}_m\right\|^2\right),\ \lambda > 0 \tag{2}$$

where the first term tries to project true neighbors (having $y^+_{nm} > 0$) close together, while the second repels all non-neighbors' projections (having $y^-_{nm} > 0$) from each other. Laplacian Eigenmaps (Belkin and Niyogi, 2003) and Locally Linear Embedding (Roweis and Saul, 2000) result from replacing the second term above with a constraint that fixes the scale of $\mathbf{Z}$, which results in an eigenproblem rather than a nonlinear optimization, but also produces more distorted embeddings. Other objectives exist, such as t-SNE (van der Maaten and Hinton, 2008), that do not separate into functions of pairs of points. Optimizing nonlinear embeddings is quite challenging, but much progress has been done recently (Carreira-Perpiñán, 2010; Vladymyrov and Carreira-Perpiñán, 2012; van der Maaten, 2013; Yang et al., 2013; Vladymyrov and Carreira-Perpiñán, 2014). Although these models were developed to produce continuous projections, they have been successfully used for binary hashing too by truncating their codes (Weiss et al., 2009; Zhang et al., 2010) or using the two-step approach of (Lin et al., 2013, 2014).

Other loss functions have been developed specifically for hashing, where now $\mathbf{z}_n$ is a b-bit vector (where binary values are in $\{-1, +1\}$). For example (see a longer list in Lin et al., 2013), for Supervised Hashing with Kernels (KSH) $L(\mathbf{z}_n, \mathbf{z}_m;\ y_{nm})$ has the form

$$(\mathbf{z}_n^T \mathbf{z}_m - b y_{nm})^2 \tag{3}$$

where y_{nm} is 1 if $\mathbf{x}_n$, $\mathbf{x}_m$ are similar and -1 if they are dissimilar. Binary Reconstructive Embeddings (Kulis and Darrell, 2009) uses $(\frac{1}{b}\left\|\mathbf{z}_n - \mathbf{z}_m\right\|^2 - y_{nm})^2$ where $y_{nm} = \frac{1}{2}\left\|\mathbf{x}_n - \mathbf{x}_m\right\|^2$. The exponential variant of SPLH (Wang et al., 2012) proposed by Lin et al. (2013) (which we call eSPLH) uses $\exp(-\frac{1}{b} y_{nm} \mathbf{z}_n^T \mathbf{z}_n)$. Our approach can be applied to any of these loss functions, though we will mostly focus on the KSH loss for simplicity. When the variables $\mathbf{Z}$ are binary, we will call these optimization problems *binary embeddings*, in analogy to the more traditional continuous embeddings for dimension reduction.

3 Learning codes and hash functions using auxiliary coordinates

The optimization of the loss $\mathcal{L}(\mathbf{h})$ in eq. (1) is difficult because of the thresholded hash function, *which appears as the argument of the loss function L*. We use the recently proposed *method of auxiliary coordinates (MAC)* (Carreira-Perpiñán and Wang, 2012, 2014), which is a meta-algorithm to construct optimization algorithms for nested functions. This proceeds in 3 stages. First, we introduce new variables (the "auxiliary coordinates") as equality constraints into the problem, with the goal of unnesting the function. We can achieve this by introducing one binary vector $\mathbf{z}_n \in \{-1, +1\}$ for each point. This transforms the original, unconstrained problem into the following, constrained problem:

$$\min_{\mathbf{h},\mathbf{Z}} \sum_{n=1}^{N} L(\mathbf{z}_n, \mathbf{z}_m;\ y_{nm}) \ \text{ s.t. } \begin{cases} \mathbf{z}_1 = \mathbf{h}(\mathbf{x}_1) \\ \cdots \\ \mathbf{z}_N = \mathbf{h}(\mathbf{x}_N) \end{cases} \tag{4}$$

which is seen to be equivalent to (1) by eliminating $\mathbf{Z}$. We recognize as the objective function the "embedding" form of the loss function, except that the "free" parameters $\mathbf{z}_n$ are in fact constrained to be the deterministic outputs of the hash function $\mathbf{h}$.

Second, we solve the constrained problem using a penalty method, such as the quadratic-penalty or augmented Lagrangian (Nocedal and Wright, 2006). We discuss here the former for simplicity. We solve the following minimization problem (unconstrained again, but dependent on μ) while progressively increasing μ, so the constraints are eventually satisfied:

$$\min \mathcal{L}_P(\mathbf{h}, \mathbf{Z}; \mu) = \sum_{n,m=1}^{N} L(\mathbf{z}_n, \mathbf{z}_m;\ y_{nm}) \ + \mu \sum_{n=1}^{N} \left\|\mathbf{z}_n - \mathbf{h}(\mathbf{x}_n)\right\|^2 \qquad \text{s.t.} \qquad \mathbf{z}_1, \dots, \mathbf{z}_N \in \{-1, 1\}^b. \tag{5}$$

The quadratic penalty $\left\|\mathbf{z}_n - \mathbf{h}(\mathbf{x}_n)\right\|^2$ is proportional to the Hamming distance between the binary vectors $\mathbf{z}_n$ and $\mathbf{h}(\mathbf{x}_n)$.

Third, we apply alternating optimization over the binary codes $\mathbf{Z}$ and the hash function parameters $\mathbf{h}$. This results in iterating the following two steps (described in detail later):

- Optimize the binary codes $\mathbf{z}_1, \dots, \mathbf{z}_N$ given $\mathbf{h}$ (hence, given the output binary codes $\mathbf{h}(\mathbf{x}_1), \dots, \mathbf{h}(\mathbf{x}_N)$ for each of the N images). This can be seen as a *regularized binary embedding*, because the projections $\mathbf{Z}$ are encouraged to be close to the hash function outputs $\mathbf{h}(\mathbf{X})$. Here, we try two different approaches (Lin et al., 2013, 2014) with some modifications.

- Optimize the hash function $\mathbf{h}$ given binary codes $\mathbf{Z}$. This reduces to training b binary classifiers using $\mathbf{X}$ as inputs and $\mathbf{Z}$ as targets.

This is very similar to the two-step (TSH) approach of Lin et al. (2013), except that the latter learns the codes $\mathbf{Z}$ in isolation, rather than given the current hash function, so iterating the two-step approach would change nothing, and it does not optimize the loss $\mathcal{L}$. More precisely, TSH corresponds to optimizing $\mathcal{L}_P$ for $\mu \to 0^+$. In practice, we start from a very small value of μ (hence, initialize MAC from the result of TSH), and increase μ slowly while optimizing $\mathcal{L}_P$, until the equality constraints are satisfied, i.e., $\mathbf{z}_n = \mathbf{h}(\mathbf{x}_n)$ for $n = 1, \dots, N$.

Fig. 1 gives the overall MAC algorithm to learn a hash function by optimizing an affinity-based loss function. We now describe the steps over $\mathbf{h}$ and $\mathbf{Z}$, and the path followed by the iterates as a function of μ.

```
input $\mathbf{X}_{D\times N} = (\mathbf{x}_1,\dots,\mathbf{x}_N)$, $\mathbf{Y}_{N\times N} = (y_{nm})$, $b \in \mathbb{N}$
Initialize $\mathbf{Z}_{b\times N} = (\mathbf{z}_1,\dots,\mathbf{z}_N) \in \{0,1\}^{bN}$
for $\mu = 0 < \mu_1 < \dots < \mu_\infty$
  for $i = 1,\dots,b$                                              h step
    $h_i \leftarrow$ fit hash function to $(\mathbf{X}, \mathbf{Z}_{\cdot i})$
  repeat                                                           Z step
    for $i = 1,\dots,b$
      $\mathbf{Z}_{\cdot i} \leftarrow$ approximate minimizer of $\mathcal{L}_P(\mathbf{h},\mathbf{Z};\mu)$ over $\mathbf{Z}_{\cdot i}$
  until no change in $\mathbf{Z}$ or maxit cycles ran
  if $\mathbf{Z} = \mathbf{h}(\mathbf{X})$ then stop
return $\mathbf{h}$, $\mathbf{Z} = \mathbf{h}(\mathbf{X})$
```

Figure 1: MAC algorithm to optimize an affinity-based loss function for binary hashing.

3.1 Stopping criterion, schedule over μ and path of optimal values

It is possible to prove that once $\mathbf{Z} = \mathbf{h}(\mathbf{X})$ after a $\mathbf{Z}$ step (regardless of the value of μ), the MAC algorithm will make no further changes to $\mathbf{Z}$ or $\mathbf{h}$, since then the constraints are satisfied. This gives us a reliable stopping criterion that is easy to check, and the MAC algorithm will stop after a finite number of iterations (see below).

It is also possible to prove that the path of minimizers of $\mathcal{L}_P$ over the continuous penalty parameter $\mu \in [0,\infty)$ is in fact discrete, with changes to $(\mathbf{Z},\mathbf{h})$ happening only at a finite number of values $0 < \mu_1 < \dots < \mu_\infty < \infty$. Based on this and on our practical experience, we have found that the following approach leads to good schedules for μ with little effort. We use exponential schedules, of the form $\mu_i = \mu_1\alpha^{i-1}$ for $i = 1,2,\dots$, so the user has to set only two parameters: the initial μ_1 and the multiplier $\alpha > 1$. We choose exponential schedules because typically the algorithm makes most progress at the beginning, and it is important to track a good minimum there. The upper value μ_∞ past which no changes occur will be reached by our exponential schedule in a finite number of iterations, and our stopping criterion will detect that. We set the multiplier to a value $1 < \alpha < 2$ that is as small as computationally convenient. If α is too small, the algorithm will take many iterations, some of which may not even change $\mathbf{Z}$ or $\mathbf{h}$ (because the path of minima is discrete). If α is too big, the algorithm will reach too quickly a stopping point, without having had time to find a better minimum. As for the initial μ_1, we estimate it by trying values (exponentially spaced) until we find a μ for which changes to $\mathbf{Z}$ from its initial value (for $\mu = 0$) start to occur. (It is also possible to find lower and upper bounds for μ_1 and μ_∞, respectively, for a particular loss function, such as KSH, eSPH or EE.) Overall, the computational time required to estimate μ_1 and α is comparable to running a few extra iterations of the MAC algorithm.

Finally, in practice we use a form of early stopping in order to improve generalization. We use a small validation set to evaluate the precision achieved by the hash function $\mathbf{h}$ along the MAC optimization. If the precision decreases over that of the previous step, we ignore the step and skip to the next value of μ. Besides helping to avoid overfitting, this saves computation, by avoid such extra optimization steps. Since the validation set is small, it provides a noisy estimate of the generalization ability at the current iterate, and this occasionally leads to skipping a valid μ value. This is not a problem because the next μ value, which is close to the one we skipped, will likely work. At some point during the MAC optimization, we do reach an overfitting region and the precision stops increasing, so the algorithm will skip all remaining μ values until it stops. In summary, using this validation procedure guarantees that the precision (in the validation set) is greater or equal than that of the initial $\mathbf{Z}$, thus resulting in a better hash function.

3.2 h step

Given the binary codes $\mathbf{z}_1, \dots, \mathbf{z}_N$, since $\mathbf{h}$ does not appear in the first term of $\mathcal{L}_P$, this simply involves finding a hash function $\mathbf{h}$ that minimizes

$$\min_{\mathbf{h}} \sum_{n=1}^{N} \left\| \mathbf{z}_n - \mathbf{h}(\mathbf{x}_n) \right\|^2 = \sum_{i=1}^{b} \min_{h_i} \sum_{n=1}^{N} (z_{ni} - h_i(\mathbf{x}_n))^2$$

where $z_{ni} \in \{-1, +1\}$ is the ith bit of the binary vector $\mathbf{z}_n$. Hence, we can find b one-bit hash functions in parallel and concatenate them into the b-bit hash function. Each of these is a binary classification problem using the number of misclassified patterns as loss. This allows us to use a regular classifier for $\mathbf{h}$, and even to use a simpler surrogate loss (such as the hinge loss), since this will also enforce the constraints eventually (as μ increases). For example, we can fit an SVM by optimizing the margin plus the slack and using a high penalty for misclassified patterns. We discuss other classifiers in the experiments.

3.3 Z step

Although the MAC technique has significantly simplified the original problem, the step over $\mathbf{Z}$ is still complex. This involves finding the binary codes given the hash function $\mathbf{h}$, and it is an NP-complete problem in Nb binary variables. Fortunately, some recent works have proposed practical approaches for this problem based on alternating optimization: a quadratic surrogate method (Lin et al., 2013), and a GraphCut method (Lin et al., 2014). In both cases, this would correspond to the first step in the two-step hashing of Lin et al. (2013).

In both the quadratic surrogate and the GraphCut method, the starting point is to apply alternating optimization over the ith bit of all points given the remaining bits are fixed for all points (for $i = 1, \dots, b$), and to solve the optimization over the ith bit approximately. We describe this next for each method. We start by describing each method in their original form (which applies to the loss function over binary codes, i.e., the first term in $\mathcal{L}_P$), and then we give our modification to make it work with our $\mathbf{Z}$ step objective (the regularized loss function over binary codes, i.e., the complete $\mathcal{L}_P$).

Solution using a quadratic surrogate method (Lin et al., 2013) This is based on the fact that any loss function that depends on the Hamming distance of two binary variables can be equivalently written as a quadratic function of those two binary variables (Lin et al., 2013). Since this is the case for every term $L(\mathbf{z}_n, \mathbf{z}_m;\ y_{nm})$ (because only the ith bit in each of $\mathbf{z}_n$ and $\mathbf{z}_m$ is free), we can write the first term in $\mathcal{L}_P$ as a binary quadratic problem. We now consider the second term (on μ) as well. (We use a similar notation as that of Lin et al., 2013.) The optimization for the ith bit can be written as:

$$\min_{\mathbf{z}_{(i)}} \sum_{n,m=1}^{N} l_i(z_{ni}, z_{mi}) + \mu \sum_{n=1}^{N} (z_{ni} - h_i(\mathbf{x}_n))^2 \tag{6}$$

where $l_i = L(z_{ni}, z_{mi}, \bar{\mathbf{z}}_n, \bar{\mathbf{z}}_m; y_{nm})$ is the loss function defined on the ith bit, z_{ni} is the ith bit of the nth point, $\bar{\mathbf{z}}_n$ is a vector containing the binary codes of the nth point except the ith bit, and $h_i(\mathbf{x}_n)$ is the ith bit of the binary code of the nth point generated by the hash function $\mathbf{h}$. Lin et al. (2013) show that $l(z_1, z_2)$ can be replaced by a binary quadratic function

$$l(z_1, z_2) = \tfrac{1}{2} z_1 z_2 \big(l^{(11)} - l^{(-11)}\big) + \text{constant} \tag{7}$$

as long as $l(1,1) = l(-1,-1) = l^{(11)}$ and $l(1,-1) = l(-1,1) = l^{(-11)}$, where $z_1, z_2 \in \{-1, 1\}$. Equation (7) helps us to rewrite the optimization (6) as the following:

$$\min_{\mathbf{z}_{(i)}} \sum_{n,m=1}^{N} \frac{1}{2} z_{ni} z_{mi} \big(l^{(11)} - l^{(-11)}\big) + \mu \sum_{n=1}^{N} (z_{ni} - h_i(\mathbf{x}_n))^2. \tag{8}$$

By defining $a_{nm} = \big(l^{(11)}_{(inm)} - l^{(-11)}_{(inm)}\big)$ as the (n, m) element of a matrix $\mathbf{A} \in \mathbb{R}^{N\times N}$ and ignoring the coefficients, we have the following optimization problem:

$$\min_{\mathbf{z}_{(i)}} \mathbf{z}_{(i)}^T \mathbf{A}\mathbf{z}_{(i)} + \mu \left\|\mathbf{z}_{(i)} - \mathbf{h}_i(\mathbf{X})\right\|^2 \qquad \text{s.t.} \qquad \mathbf{z}_{(i)} \in \{-1, +1\}^N$$

where $\mathbf{h}_i(\mathbf{X}) = (h_i(\mathbf{x}_1), \dots, h_i(\mathbf{x}_N))^T$ is a vector of length N (one bit per data point). Both terms in the above minimization are quadratic on binary variables. This is still an NP-complete problem (except in special cases), and we approximate it by relaxing it to a continuous quadratic program (QP) over $\mathbf{z}_{(i)} \in [-1, 1]^N$ and binarizing its solution. In general, the matrix $\mathbf{A}$ is not positive definite and the relaxed QP is not convex, so we need an initialization. (However, the term on μ adds $\mu\mathbf{I}$ to $\mathbf{A}$, so even if $\mathbf{A}$ is not positive definite, $\mathbf{A} + \mu\mathbf{I}$ will be positive definite for large enough μ, and the QP will be convex.) We construct an initialization by converting the binary QP into a binary eigenproblem:

$$\min_{\boldsymbol{\alpha}} \boldsymbol{\alpha}^T \mathbf{B}\boldsymbol{\alpha} \qquad \text{s.t.} \qquad \alpha_0 = 1, \quad \mathbf{z}_{(i)} \in \{-1, 1\}^N, \quad \boldsymbol{\alpha} = \begin{pmatrix} \mathbf{z}_{(i)} \\ \alpha_0 \end{pmatrix}, \quad \mathbf{B} = \begin{pmatrix} \mathbf{A} & -\frac{\mu}{2}\mathbf{h}_i(\mathbf{X}) \\ -\frac{\mu}{2}\mathbf{h}_i(\mathbf{X})^T & 0 \end{pmatrix}. \tag{9}$$

To solve this problem we use spectral relaxation, where the constraints $\mathbf{z}_{(i)} \in \{-1, +1\}^N$ and $z_{i+1} = 1$ are relaxed to $\|\boldsymbol{\alpha}\| = N + 1$. The solution to this problem is the eigenvector corresponding to the smallest eigenvalue of $\mathbf{B}$. We use the truncated eigenvector as the initialization for minimizing the relaxed, bound-constrained QP:

$$\min_{\mathbf{z}_{(i)}} \mathbf{z}_{(i)}^T \mathbf{A}\mathbf{z}_{(i)} + \mu \left\|\mathbf{z}_{(i)} - \mathbf{h}_i(\mathbf{X})\right\|^2 \qquad \text{s.t.} \qquad \mathbf{z}_{(i)} \in [-1, 1]^N.$$

which we solve using L-BFGS-B (Zhu et al., 1997).

As noted above, the $\mathbf{Z}$ step is an NP-complete problem in general, so we cannot expect to find the global optimum. It is even possible that the approximate solution could increase the objective over the previous iteration's $\mathbf{Z}$ (this is likely to happen as the overall MAC algorithm converges). If that occurs, we simply skip the update, in order to guarantee that we decrease monotonically on $\mathcal{L}_P$, and avoid oscillating around a minimum.

Solution using a GraphCut algorithm (Lin et al., 2014) To optimize over the ith bit (given all the other bits are fixed), we have to minimize eq. (8). In general, this is an NP-complete problem over N bits (the ith bit for each image), with the form of a quadratic function on binary variables. We can apply the GraphCut algorithm (Boykov and Kolmogorov, 2003, 2004; Kolmogorov and Zabih, 2003), as proposed by the FastHash algorithm of Lin et al. (2014). This proceeds as follows. First, we assign all the data points to different, possibly overlapping groups (blocks). Then, we minimize the objective function over the binary codes of the same block, while all the other binary codes are fixed, then proceed with the next block, etc. (that is, we do alternating optimization of the bits over the blocks). Specifically, to optimize over the bits in block $\mathcal{B}$, we define $a_{nm} = \big(l^{(11)}_{(inm)} - l^{(-11)}_{(inm)}\big)$ and, ignoring the constants, we can rewrite equation (8) as:

$$\min_{\mathbf{z}_{(i,\mathcal{B})}} \sum_{n\in\mathcal{B}} \sum_{m\in\mathcal{B}} a_{nm} z_{ni} z_{mi} + 2 \sum_{n\in\mathcal{B}} \sum_{m\notin\mathcal{B}} a_{nm} z_{ni} z_{mi} - \mu \sum_{n\in\mathcal{B}} z_{ni} h_i(\mathbf{x}_n).$$

We then rewrite this equation in the standard form for the GraphCut algorithm:

$$\min_{\mathbf{z}_{(i,\mathcal{B})}} \sum_{n\in\mathcal{B}} \sum_{m\in\mathcal{B}} v_{nm} z_{ni} z_{mi} + \sum_{n\in\mathcal{B}} u_{nm} z_{ni}$$

where $v_{nm} = a_{nm}$, $u_{nm} = 2\sum_{m\notin\mathcal{B}} a_{nm} z_{mi} - \mu h_i(\mathbf{x}_n)$. To minimize the objective function using the GraphCut algorithm, the blocks have to define a submodular function. For the objective functions that we explained in the paper, this can be easily achieved by putting points with the same label in one block (Lin et al., 2014 give a simple proof of this).

Unlike in the quadratic surrogate method, using the GraphCut algorithm with alternating optimization on blocks defining submodular functions is guaranteed to find a $\mathbf{Z}$ that has a lower or equal objective value that the initial one, and therefore to decrease monotonically $\mathcal{L}_P$.

4 Experiments

We have tested our framework with several combinations of loss function, hash function, number of bits, datasets, and comparing with several state-of-the-art hashing methods (appendix A contains additional experiments). We report a representative subset to show the flexibility of the approach. We use the KSH (3) (Liu et al., 2012) and eSPLH (Wang et al., 2012) loss functions. We test quadratic surrogate and GraphCut methods for the $\mathbf{Z}$ step in MAC. As hash functions (for each bit), we use linear SVMs (trained with LIBLINEAR; Fan et al., 2008) and kernel SVMs[1]

We use the following labeled datasets (all using the Euclidean distance in feature space): (1) CIFAR (Krizhevsky, 2009) contains 60 000 images in 10 classes. We use $D = 320$ GIST features (Oliva and Torralba, 2001) from each image. We use 58 000 images for training and 2 000 for test. (2) Infinite MNIST (Loosli et al., 2007). We generated, using elastic deformations of the original MNIST handwritten digit dataset, 1 000 000 images for training and 2 000 for test, in 10 classes. We represent each image by a $D = 784$ vector of raw pixels. Because of the computational cost of affinity-based methods, previous work has used training sets limited to a few thousand points (Kulis and Darrell, 2009; Norouzi and Fleet, 2011; Liu et al., 2012; Lin et al., 2013). We train the hash functions in a subset of 10 000 points of the training set, and report precision and recall by searching for a test query on the entire dataset (the base set).

We report precision and precision/recall for the test set queries using as ground truth (set of true neighbors in original space) all the training points with the same label. In precision curves, the retrieved set contains the k nearest neighbors of the query point in the Hamming space. We report precision for different values of k to test the robustness of different algorithms. In precision/recall curves, the retrieved set contains the points inside Hamming distance r of the query point. These curves show the precision and recall at different Hamming distances $r = 0$ to $r = L$. We report zero precision when there is no neighbor inside Hamming distance r of a query. This happens most of the time when L is large and r is small. In most of our precision/recall curves, the precision drops significantly for very small and very large values of r. For small values of r, this happens because most of the query points do not retrieve any neighbor. For large values of r, this happens because the number of retrieved points becomes very large.

The main comparison point are the quadratic surrogate and GraphCut methods of Lin et al. (2013, 2014), which we denote in this section as *quad* and *cut*, respectively, regardless of the hash function that fits the resulting codes. Correspondingly, we denote the MAC version of these as *MACquad* and *MACcut*, respectively. We use the following schedule for the penalty parameter μ in the MAC algorithm (regardless of the hash function type or dataset). We initialize $\mathbf{Z}$ with $\mu = 0$, i.e., the result of *quad* or *cut*. Starting from $\mu_1 = 0.3$ (*MACcut*) or 0.01 (*MACquad*), we multiply μ by 1.4 after each iteration ($\mathbf{Z}$ and $\mathbf{h}$ step).

Our experiments show that the MAC algorithm indeed finds hash functions with a significantly and consistently lower objective function value than rounding or two-step approaches (in particular, *cut* and *quad*); and that it outperforms other state-of-the-art algorithms on different datasets, with *MACcut* beating *MACquad* most of the time. The improvement in precision makes using MAC well worth the relatively small extra runtime and minimal additional implementation effort it requires. In all our plots, the vertical arrows indicate the improvement of *MACcut* over *cut* and of *MACquad* over *quad*.

4.1 The MAC algorithm finds better optima

The goal of this paper is not to introduce a new affinity-based loss or hash function, but to describe a generic framework to construct algorithms that optimize a given combination thereof. We illustrate its effectiveness here with the CIFAR dataset, with different sizes of retrieved neighbor sets, and using 16 to 48 bits. We optimize two affinity-based loss functions (KSH from eq. (3) and eSPLH), and two hash functions (linear and kernel SVM). In all cases, the MAC algorithm achieves a better hash function both in terms of the loss and of the precision/recall. We compare 4 ways of optimizing the loss function: *quad* (Lin et al., 2013), *cut* (Lin et al., 2014), *MACquad* and *MACcut*.

For each point $\mathbf{x}_n$ in the training set, we use $\kappa_+ = 100$ positive (similar) and $\kappa_- = 500$ negative

[1] To train a kernel SVM, we use 500 radial basis functions with centers given by a random subset of the training points, and apply a linear SVM to their output. Computationally, this is fast because we can use a constant Gram matrix. Using as hash function a kernel SVM trained with LIBSVM gave similar results, but is much slower because the support vectors change when the labels change. We set the RBF bandwidth to the average Euclidean distance of the first 300 points.

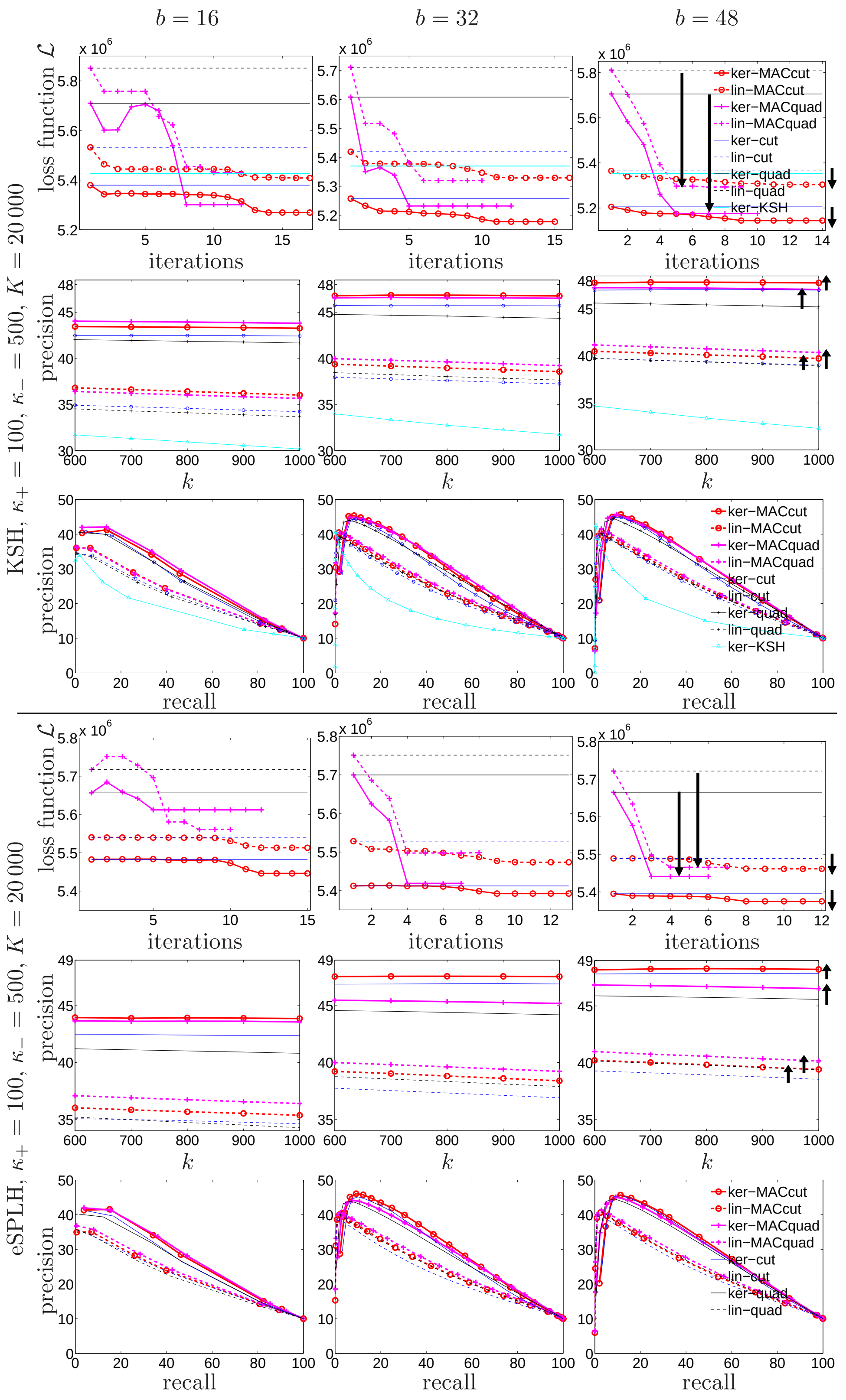

Figure 2: KSH (top panel) and eSPLH (bottom panel) loss functions on CIFAR dataset, using $b = 16$ to 48 bits. The rows in each panel show the value of the loss function $\mathcal{L}$, the precision for k retrieved points and the precision/recall (at different Hamming distances).

(dissimilar) neighbors, chosen at random to have the same or a different label as $\mathbf{x}_n$, respectively. Fig. 2(top panel) shows the KSH loss function for all the methods (including the original KSH method in Liu et al., 2012) over iterations of the MAC algorithm (KSH, *quad* and *cut* do not iterate), as well as precision and recall. It is clear that *MACcut* (red lines) and *MACquad* (magenta lines) reduce the loss function more than *cut* (blue lines) and *quad* (black lines), respectively, as well as the original KSH algorithm (cyan), in all cases: type of hash function (linear: dashed lines, kernel: solid lines) and number of bits $b = 16$ to 48. Hence, applying MAC is always beneficial. Reducing the loss nearly always translates into better precision and recall (with a larger gain for linear than for kernel hash functions, usually). The gain of *MACcut*/*MACquad* over *cut*/*quad* is significant, often comparable to the gain obtained by changing from the linear to the kernel hash function within the same algorithm.

We usually find *cut* outperforms *quad* (in agreement with Lin et al., 2014), and correspondingly *MACcut* outperforms *MACquad*. Interestingly, *MACquad* and *MACcut* end up being very similar even though they started very differently. This suggests it is not crucial which of the two methods to use in the MAC $\mathbf{Z}$ step, although we still prefer *cut*, because it usually produces somewhat better optima. Finally, fig. 2(bottom panel) shows the *MACcut* results using the eSPLH loss. All settings are as in the first KSH experiment. As before, *MACcut* outperforms *cut* in both loss function and precision/recall using either a linear or a kernel SVM.

4.2 Why does MAC learn better hash functions?

In both the two-step and MAC approaches, the starting point are the "free" binary codes obtained by minimizing the loss over the codes without them being the output of a particular hash function. That is, minimizing (4) without the "$\mathbf{z}_n = \mathbf{h}(\mathbf{x}_n)$" constraints:

$$\min_{\mathbf{Z}} E(\mathbf{Z}) = \sum_{n=1}^{N} L(\mathbf{z}_n, \mathbf{z}_m;\ y_{nm}),\ \mathbf{z}_1, \dots, \mathbf{z}_N \in \{-1, +1\}^b. \tag{10}$$

The resulting free codes try to achieve good precision/recall independently of whether a hash function can actually produce such codes. Constraining the codes to be realizable by a specific family of hash functions (say, linear), means the loss $E(\mathbf{Z})$ will be larger than for free codes. How difficult is it for a hash function to produce the free codes? Fig. 3 plots the loss function for the free codes, the two-step codes from *cut*, and the codes from *MACcut*, for both linear and kernel hash functions in the same experiment as in fig. 2. It is clear that the free codes have a very low loss $E(\mathbf{Z})$, which is far from what a kernel function can produce, and even farther from what a linear function can produce. Both of these are relatively smooth functions that cannot represent the presumably complex structure of the free codes. This could be improved by using a very flexible hash function (e.g. using a kernel function with many centers), which could better approximate the free codes, but 1) a very flexible function would likely not generalize well, and 2) we require fast hash functions for fast retrieval anyway. Given our linear or kernel hash functions, what the two-step *cut* optimization does is fit the hash function directly to the free codes. This is not guaranteed to find the best hash function in terms of the original problem (1), and indeed it produces a pretty suboptimal function. In contrast, MAC gradually optimizes both the codes and the hash function so they eventually match, and finds a better hash function for the original problem (although it is still not guaranteed to find the globally optimal function of problem (1), which is NP-complete).

Fig. 4 illustrates this conceptually. It shows the space of all possible binary codes, the contours of $E(\mathbf{Z})$ (green) and the set of codes that can be produced by (say) linear hash functions $\mathbf{h}$ (gray), which is the feasible set $\{\mathbf{Z} \in \{-1, +1\}^{b\times N}\colon\ \mathbf{Z} = \mathbf{h}(\mathbf{X})$ for linear $\mathbf{h}\}$. The two-step codes "project" the free codes onto the feasible set, but these are not the codes for the optimal hash function $\mathbf{h}$.

4.3 Runtime

The runtime per iteration for our 10 000-point training sets with $b = 48$ bits and $\kappa_+ = 100$ and $\kappa_- = 500$ neighbors in a laptop is 2' for both *MACcut* and *MACquad*. They stop after 10–20 iterations. Each iteration is comparable to a single *cut* or *quad* run, since the $\mathbf{Z}$ step dominates the computation. The iterations after the first one are faster because they are warm-started.

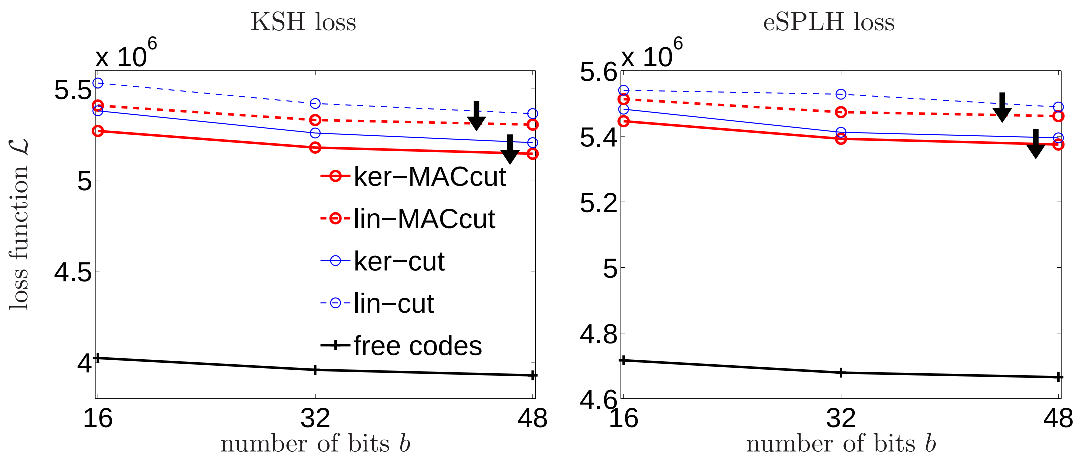

Figure 3: Like fig. 2 but showing the value of the error function $E(\mathbf{Z})$ of eq. (10) for the "free" binary codes, and for the codes produced by the hash functions learned by *cut* (the two-step method) and *MACcut*, with linear and kernel hash functions.

4.4 Comparison with binary hashing methods

Fig. 5 shows results on CIFAR and Infinite MNIST. We create affinities y_{nm} for all methods using the dataset labels as before, with $\kappa_+ = 100$ similar neighbors and $\kappa_- = 500$ dissimilar neighbors. We compare *MACquad* and *MACcut* with Two-Step Hashing (*quad*) (Lin et al., 2013), FastHash (*cut*) (Lin et al., 2014), Hashing with Kernels (KSH) (Liu et al., 2012), Iterative Quantization (ITQ) (Gong et al., 2013), Binary Reconstructive Embeddings (BRE) (Kulis and Darrell, 2009) and Self-Taught Hashing (STH) (Zhang et al., 2010). *MACquad*, *MACcut*, *quad* and *cut* all use the KSH loss function (3). The results show that *MACcut* (and *MACquad*) generally outperform all other methods, often by a large margin, in nearly all situations (dataset, number of bits, size of retrieved set). In particular, *MACcut* and *MACquad* are the only ones to beat ITQ, as long as one uses sufficiently many bits.

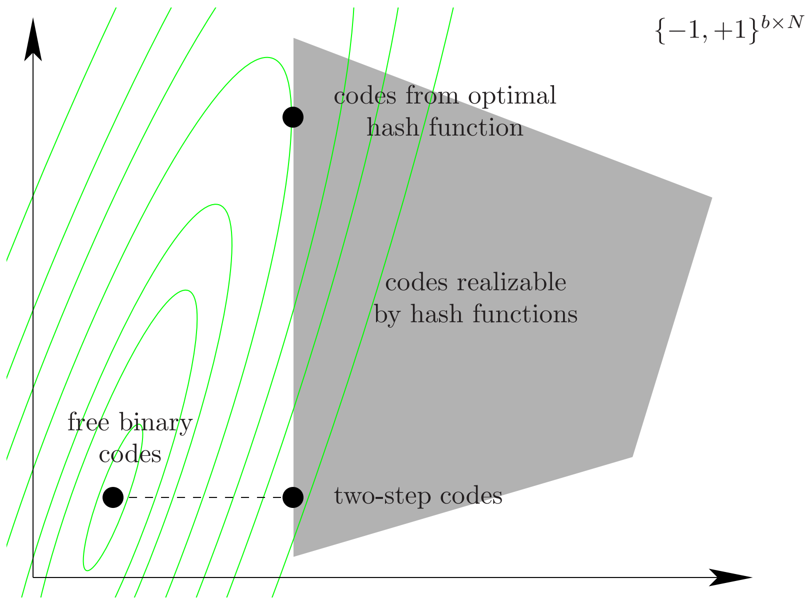
$\{-1,+1\}^{b\times N}$
codes from optimal hash function
codes realizable by hash functions
free binary codes
two-step codes

Figure 4: Illustration of free codes, two-step codes and optimal codes realizable by a hash function, in the space $\{-1,+1\}^{b\times N}$.

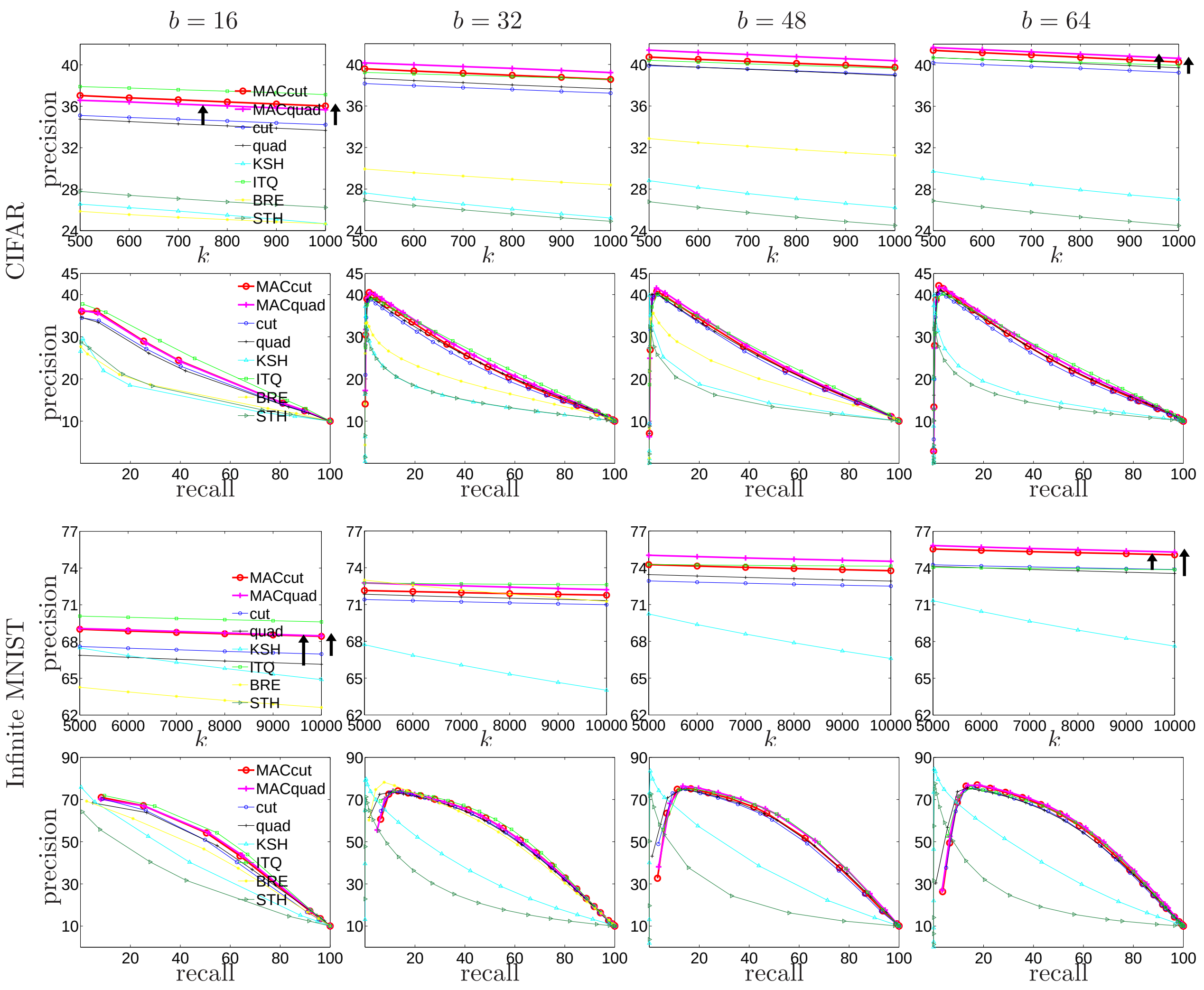

Figure 5: Comparison with binary hashing methods on CIFAR (top panel) and Infinite MNIST (bottom panel), using a linear hash function, using $b = 16$ to 64 bits. The rows in each panel show the precision for k retrieved points, for a range of k, and the precision/recall at different Hamming distances.

5 Discussion

Two-step approaches vs the MAC algorithm for affinity-based loss functions The two-step approach of Two-Step Hashing (Lin et al., 2013) and FastHash (Lin et al., 2014) is a significant advance in finding good codes for binary hashing, but it also causes a maladjustment between the codes and the hash function, since the codes were learned without knowledge of what hash function would use them. Ignoring the interaction between the loss and the hash function limits the quality of the results. For example, a linear hash function will have a harder time than a nonlinear one at learning such codes. In our algorithm, this tradeoff is enforced gradually (as μ increases) in the $\mathbf{Z}$ step as a regularization term (eq. (5)): it finds the best codes according to the loss function, but makes sure they are close to being realizable by the current hash function. Our experiments demonstrate that significant, consistent gains are achieved in both the loss function value and the precision/recall in image retrieval over the two-step approach.

A similar, well-known situation arises in feature selection for classification (Kohavi and John, 1998). The best combination of classifier and features will result from jointly minimizing the classification error with respect to both classifier and features (the "wrapper" approach), rather than first selecting features according to some criterion and then using them to learn a particular classifier (the "filter" approach). From this point

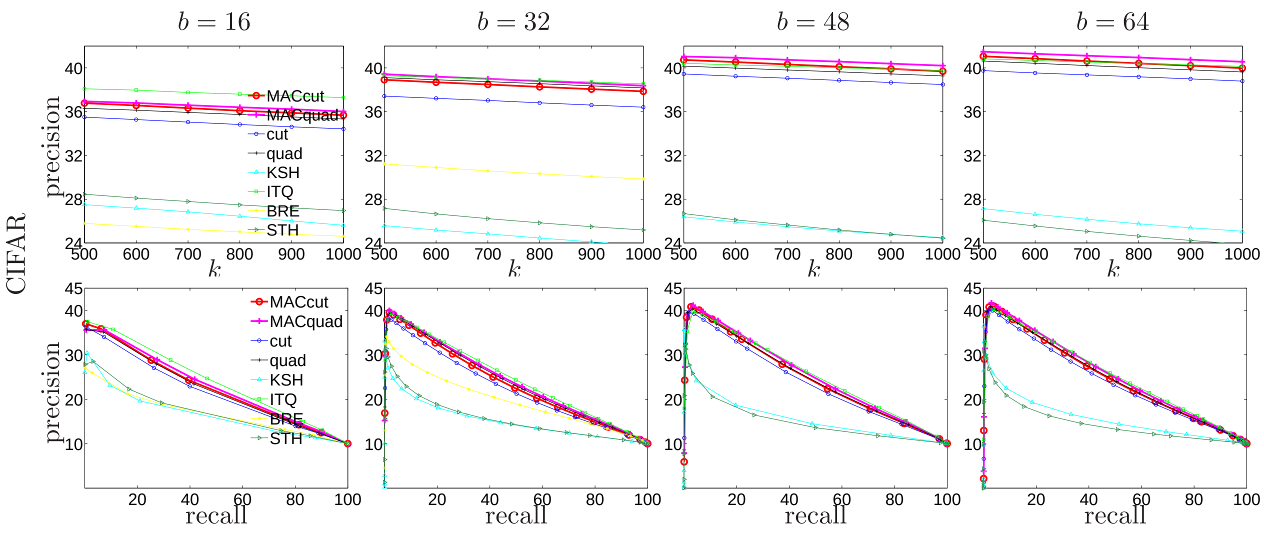

Figure 6: As in fig. 5 but using the cosine similarity instead of the Euclidean distance to find neighbors (i.e., all the points are centered and normalized before training and testing), on CIFAR.

of view, the two-step approaches of (Lin et al., 2013, 2014) are filter approaches that first optimize the loss function over the codes $\mathbf{Z}$ (equivalently, optimize $\mathcal{L}_P$ with $\mu = 0$) and then fit the hash function $\mathbf{h}$ to those codes. Any such filter approach is then equivalent to optimizing $\mathcal{L}_P$ over $(\mathbf{Z}, \mathbf{h})$ for $\mu \to 0^+$.

The method of auxiliary coordinates algorithmically decouples (within each iteration) the two elements that make up a binary hashing model: the hash function and the loss function. Both elements act in combination to produce a function that maps input patterns to binary codes so that they represent neighborhood in input space, but they play distinct roles. The hash function role is to map input patterns to binary codes. The loss function role is to assign binary codes to input patterns in order to preserve neighborhood relations, regardless of how easy it is for a mapping to produce such binary codes. By itself, the loss function would produce a nonparametric hash function for the training set with the form of a table of (image,code) pairs. However, the hash function and the loss function cannot act independently, because the objective function depends on both. The optimal combination of hash and loss is difficult to obtain, because of the nonlinear and discrete nature of the objective. Several previous optimization attempts for binary hashing first find codes that optimize the loss, and then fit a hash function to them, thus imposing a strict, suboptimal separation between loss function and hash function. In MAC, both elements are decoupled within each iteration, while still optimizing the correct objective: the step over the hash function does not involve the loss, and the step over the codes does not involve the hash function, but both are iterated. The connection between both steps occurs through the auxiliary coordinates, which are the binary codes themselves. The penalty regularizes the loss so that its optimal codes are progressively closer to what a hash function from the given class (e.g. linear) can achieve.

What is the best type of hash function to use? The answer to this is not unique, as it depends on application-specific factors: quality of the codes produced (to retrieve the correct images), time to compute the codes on high-dimensional data (since, after all, the reason to use binary hashing is to speed up retrieval), ease of implementation within a given hardware architecture and software libraries, etc. Our MAC framework facilitates considerably this choice, because training different types of hash functions simply involves reusing an existing classification algorithm within the $\mathbf{h}$ step, with no changes to the $\mathbf{Z}$ step.

In terms of runtime, the resulting MAC algorithm is not much slower than the two-step approach; it is comparable to iterating the latter a few times. Besides, since all iterations except the first are warm-started, the average cost of one iteration is lower than for the two-step approach.

Finally, note that the method of auxiliary coordinates can be used also to learn an out-of-sample mapping for a *continuous embedding* (Carreira-Perpiñán and Vladymyrov, 2015), such as the elastic embedding (Carreira-Perpiñán, 2010) or t-SNE (van der Maaten and Hinton, 2008)—rather than to learn hash functions for a discrete embedding, as is our case in binary hashing. The resulting MAC algorithm optimizes over the out-of-sample mapping and the auxiliary coordinates (which are the data points' low-dimensional projections), by alternating two steps. One step optimizes the out-of-sample mapping that projects high-dimensional points to the continuous, latent space, given the auxiliary coordinates $\mathbf{Z}$. This is a regression problem, while in binary hashing this is a classification problem (per hash function). The other step optimizes the auxiliary coordinates $\mathbf{Z}$ given the mapping and is a regularized continuous embedding problem. Both steps can be solved using existing algorithms. In particular, solving the $\mathbf{Z}$ step can be done efficiently with large datasets by using N-body methods and efficient optimization techniques (Carreira-Perpiñán, 2010; Vladymyrov and Carreira-Perpiñán, 2012; van der Maaten, 2013; Yang et al., 2013; Vladymyrov and Carreira-Perpiñán, 2014). In binary hashing the $\mathbf{Z}$ step is a combinatorial optimization and, at present, far more challenging to solve. However, with continuous embeddings one must drive the penalty parameter μ to infinity for the constraints to be satisfied and so the solution follows a continuous path over $\mu \in \mathbb{R}$, while with binary hashing the solution follows a discretized, piecewise path which terminates at a finite value of μ.

Binary autoencoder vs affinity-based loss, trained with MAC The method of auxiliary coordinates has also been applied in the context of binary hashing to a different objective function, the *binary autoencoder (BA)* (Carreira-Perpiñán and Raziperchikolaei, 2015):

$$E_{\mathrm{BA}}(\mathbf{h},\mathbf{f}) = \sum_{n=1}^{N} \left\| \mathbf{x}_n - \mathbf{f}(\mathbf{h}(\mathbf{x}_n)) \right\|^2 \tag{11}$$

where $\mathbf{h}$ is the hash function, or encoder (which outputs binary values), and $\mathbf{f}$ is a decoder. (ITQ, Gong et al., 2013, can be seen as a suboptimal way to optimize this.) As with the affinity-based loss function, the MAC algorithm alternates between fitting the hash function (and the decoder) given the codes, and optimizing over the codes. However, *in the binary autoencoder the optimization over the codes decouples over every data point* (since the objective function involves one term per data point). This has an important computational advantage in the $\mathbf{Z}$ step: rather than having to solve one large optimization problem $\{\mathbf{z}_1,\dots,\mathbf{z}_N\}$ over Nb binary variables, it has to solve N small optimization problems $\{\mathbf{z}_1\},\dots,\{\mathbf{z}_N\}$ each over b variables, which is much faster and easier to solve (since b is relatively small in practice), and to parallelize. Also, the BA objective does not require any neighborhood information (e.g. the affinity between pairs of neighbors) and scales linearly with the dataset. Computing these affinity values, or even finding pairs of neighbors in the first place, is computationally costly. For these reasons, the BA can scale to training on larger datasets than affinity-based loss functions.

The BA objective function does have the disadvantage of being less directly related to the goals that are desirable from an information retrieval point of view, such as precision and recall. Neighborhood relations are only indirectly preserved by autoencoders (Carreira-Perpiñán and Raziperchikolaei, 2015), whose direct aim is to reconstruct its inputs and thus to learn the data manifold (imperfectly, because of the binary projection layer). Affinity-based loss functions of the form (1) allow the user to specify more complex neighborhood relations, for example based on class labels, which may significantly differ from the actual distances in image feature space. Still, finding more efficient and scalable optimization methods for binary embeddings (in the $\mathbf{Z}$ step of the MAC algorithm), that are able to handle larger numbers of training and neighbor points, would improve the quality of the loss function. This is an important topic of future research.

6 Conclusion

We have proposed a general framework for optimizing binary hashing using affinity-based loss functions. It improves over previous, two-step approaches based on learning binary codes first and then learning the hash function. Instead, it optimizes jointly over the binary codes and the hash function in alternation, so that the binary codes eventually match the hash function, resulting in a better local optimum of the affinity-based loss. This was possible by introducing auxiliary variables that conditionally decouple the codes from the hash function, and gradually enforcing the corresponding constraints. Our framework makes it easy to design an optimization algorithm for a new choice of loss function or hash function: one simply reuses existing software that optimizes each in isolation. The resulting algorithm is not much slower than the suboptimal two-step approach—it is comparable to iterating the latter a few times—and well worth the improvement in precision/recall.

The step over the hash function is essentially a solved problem if using a classifier, since this can be learned in an accurate and scalable way using machine learning techniques. The most difficult and time-consuming part in our approach is the optimization over the binary codes, which is NP-complete and involves many binary variables and terms in the objective. Although some techniques exist (Lin et al., 2013, 2014) that produce practical results, designing algorithms that reliably find good local optima and scale to large training sets is an important topic of future research.

Another direction for future work involves learning more sophisticated hash functions that go beyond mapping image features onto output binary codes using simple classifiers such as SVMs. This is possible because the optimization over the hash function parameters is confined to the **h** step and takes the form of a supervised classification problem, so we can apply an array of techniques from machine learning and computer vision. For example, it may be possible to learn image features that work better with hashing than standard features such as SIFT, or to learn transformations of the input to which the binary codes should be invariant, such as translation, rotation or alignment.

A Additional experiments

A.1 Unsupervised dataset

Although affinity-based hashing is intended to work with supervised datasets, it can also be used with unsupervised ones, and our MAC approach applies just as well. We use the SIFT1M dataset (Jégou et al., 2011), which contains $N = 1\,000\,000$ training high-resolution color images and 10 000 test images, each represented by $D = 128$ SIFT features. The experiments and conclusions are generally the same as with supervised datasets, with small differences in the settings of the experiments. In order to construct an affinity-based objective function, we define neighbors as follows. For each point in the training set we use the $\kappa_+ = 100$ nearest neighbors as positive (similar) neighbors, and $\kappa_- = 500$ points chosen randomly among the remaining points as negative (dissimilar) neighbors. We report precision and precision/recall for the test set queries using as ground truth (set of true neighbors in original space) the K nearest neighbors in unsupervised datasets, and all the training points with the same label in supervised datasets.

Fig. 7 shows results using KSH and eSLPH loss functions, respectively, with different sizes of retrieved neighbor sets and using 8 to 32 bits. As with the supervised datasets, it is clear that the MAC algorithm finds better optima and that *MACcut* is generally better than *MACquad*. Fig. 8 shows one case, using $\kappa_+ = 50$, $\kappa_- = 1\,000$ and $K = 10\,000$ (1% of the base set), where *quad* outperforms *cut* and correspondingly *MACquad* outperforms *MACcut*, although both MAC results are very close, particularly in precision and recall.

Fig. 10 shows results comparing with binary hashing methods. All methods are trained on a subset of 5 000 points. We consider two types of methods. In the first type, we create pseudolabels for each point and then apply supervised methods as in CIFAR (in particular, *cut*/*quad* and *MACcut*/*MACquad*, using the KSH loss function). The pseudolabels y_{nm} for each training point $\mathbf{x}_n$ are obtained by declaring as similar points its $\kappa_+ = 100$ true nearest neighbors and as dissimilar points a random subset of $\kappa_- = 500$ points among the remaining points. In the second type, we use purely unsupervised methods (not based on similar/dissimilar affinities): thresholded PCA (tPCA), Iterative Quantization (ITQ) (Gong et al., 2013), Binary Autoencoder (BA) (Carreira-Perpiñán and Raziperchikolaei, 2015), Spectral Hashing (SH) (Weiss et al., 2009), Anchor-Graph Hashing (AGH) (Liu et al., 2011), and Spherical Hashing (SPH) (Heo et al., 2012). The results are again in general agreement with the conclusions in the main paper.

Comparison using code utilization Fig. 12 shows the results (for all methods on SIFT1M) in effective number of bits b_{eff}. This is a measure of code utilization of a hash function introduced by Carreira-Perpiñán and Raziperchikola (2015), defined as the entropy of the code distribution. That is, given the N codes $\mathbf{z}_1, \dots, \mathbf{z}_N \in \{0,1\}^b$ for the training set, we consider them as samples of a distribution over the 2^b possible codes. The entropy of this distribution, measured in bits, is between 0 (when all N codes are equal) and $\min(b, \log_2 N)$ (when all N codes are distributed as uniformly as possible). We do the same for the test set. Although code utilization correlates to some extent with precision/recall when ranking different methods, a large b_{eff} does not guarantee a good hash function, and indeed, tPCA (which usually achieves a low precision compared to the state-of-the-art) typically achieves the largest b_{eff}; see the discussion in Carreira-Perpiñán and Raziperchikolaei (2015).

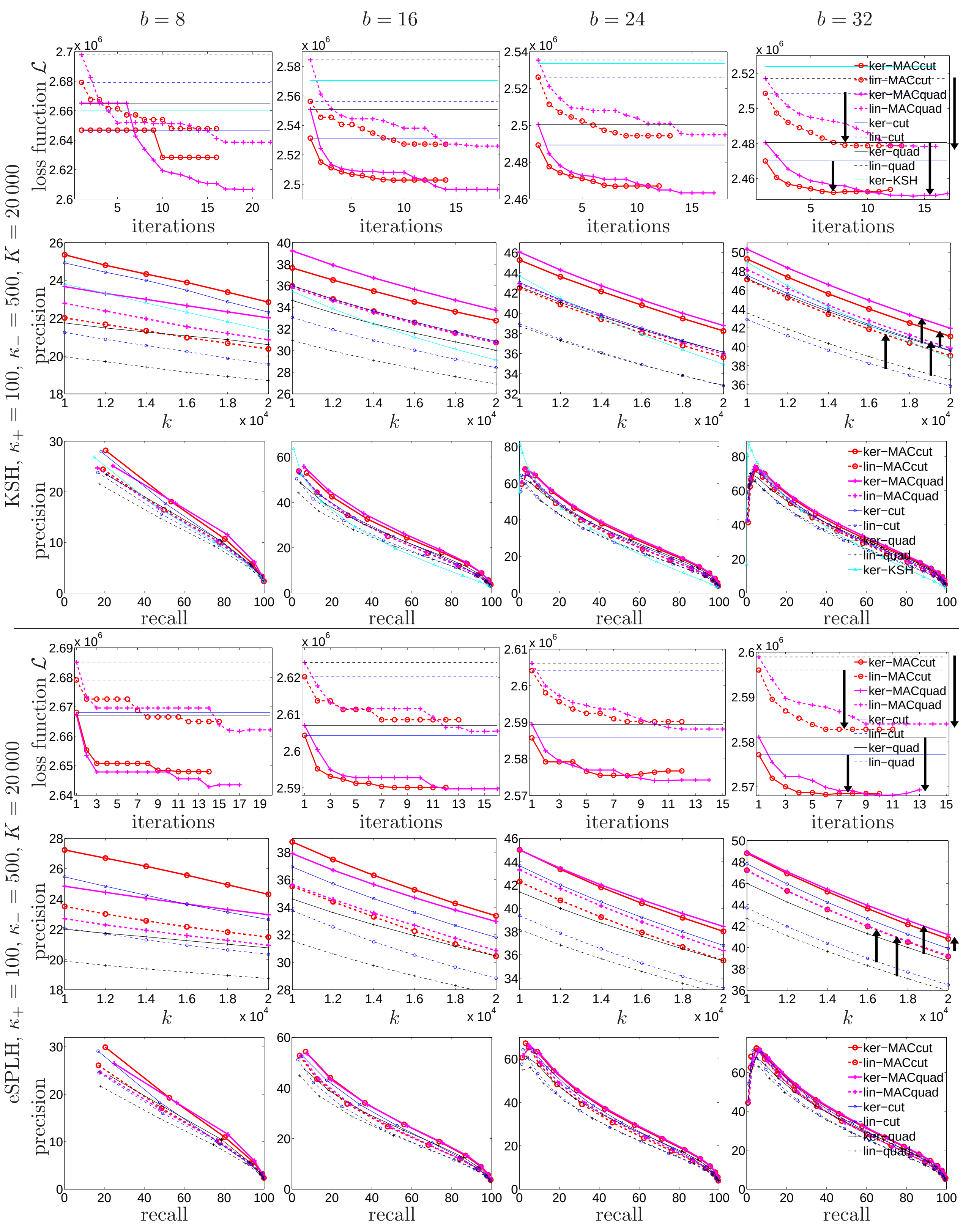

Figure 7: Like fig. 2 but on SIFT1M dataset, for the KSH (top panel) and eSPLH (bottom panel) loss functions. The rows show the value of the loss function $\mathcal{L}$, the precision (for a number of retrieved points k) and the precision/recall (at different Hamming distances), using $b = 8$ to 32 bits.

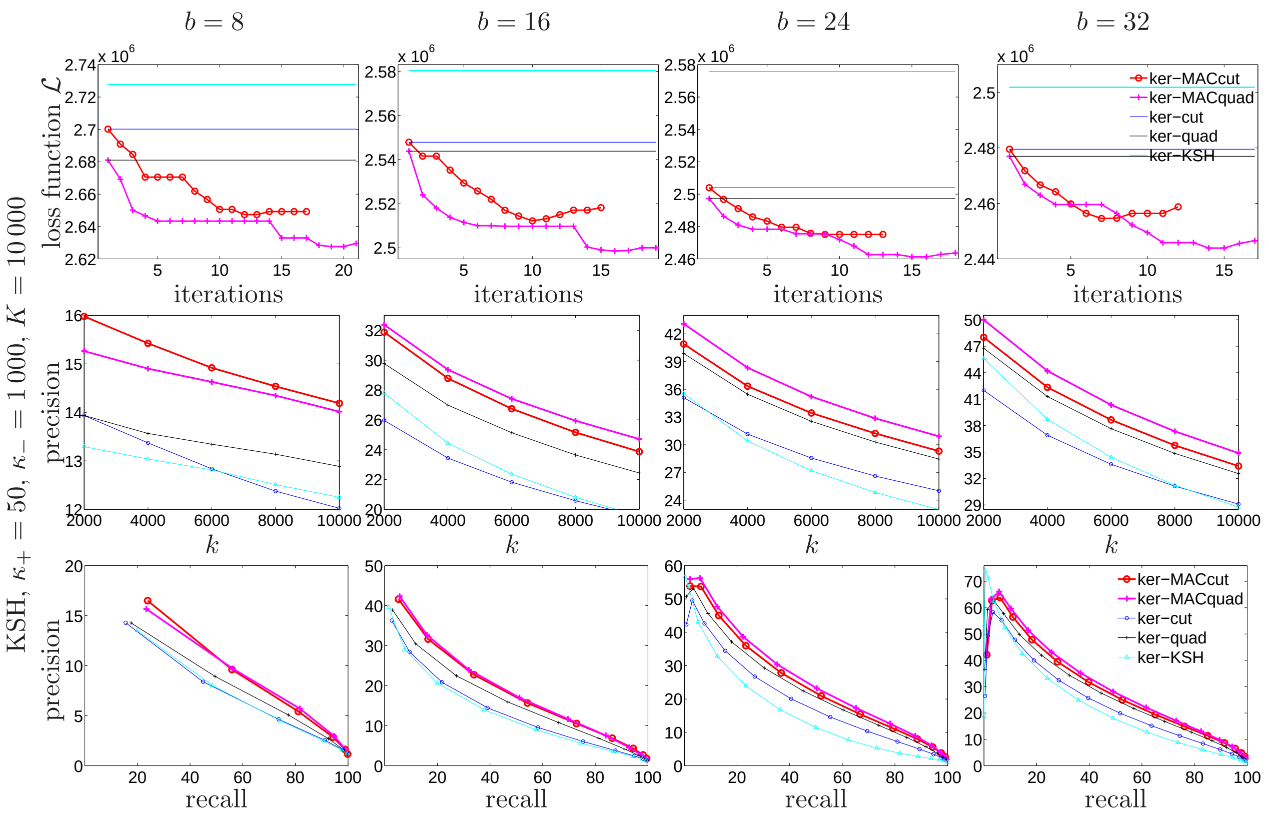

Figure 8: Like the top panel of fig. 7 (KSH loss) but with $\kappa_+ = 50$, $\kappa_- = 1\,000$, $K = 10\,000$.

However, a large b_{eff} does indicate a better use of the available codes (and fewer collisions if $N < 2^b$), and b_{eff} has the advantage over precision/recall that it does not depend on any user parameters (such as ground truth size or retrieved set size), so we can compare all binary hashing methods with a single number b_{eff} (for a given number of bits b). It is particularly useful to compare methods that are optimizing the same objective function. With this in mind, we can compare *MACcut* with *cut* and *MACquad* with *quad* because these pairs of methods optimize the same objective function.

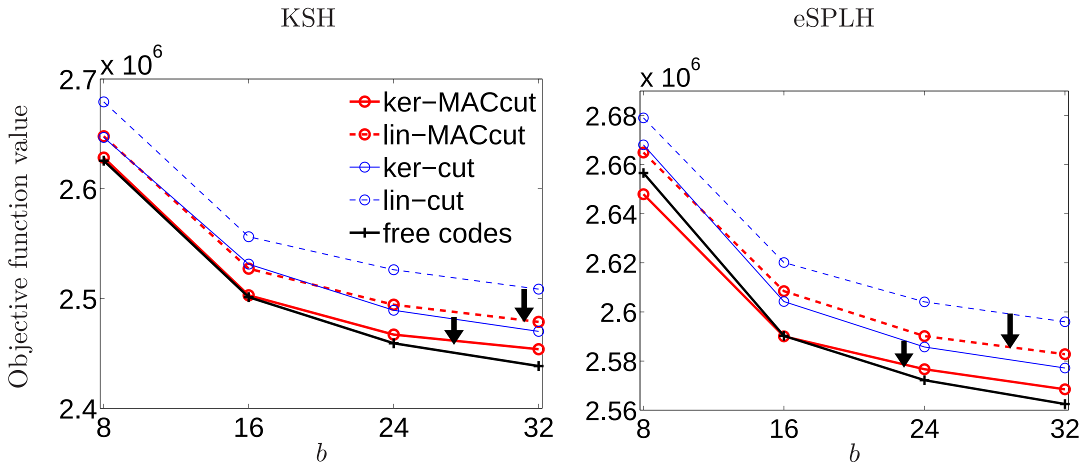

Figure 9: Like fig. 3 but for the SIFT1M dataset.

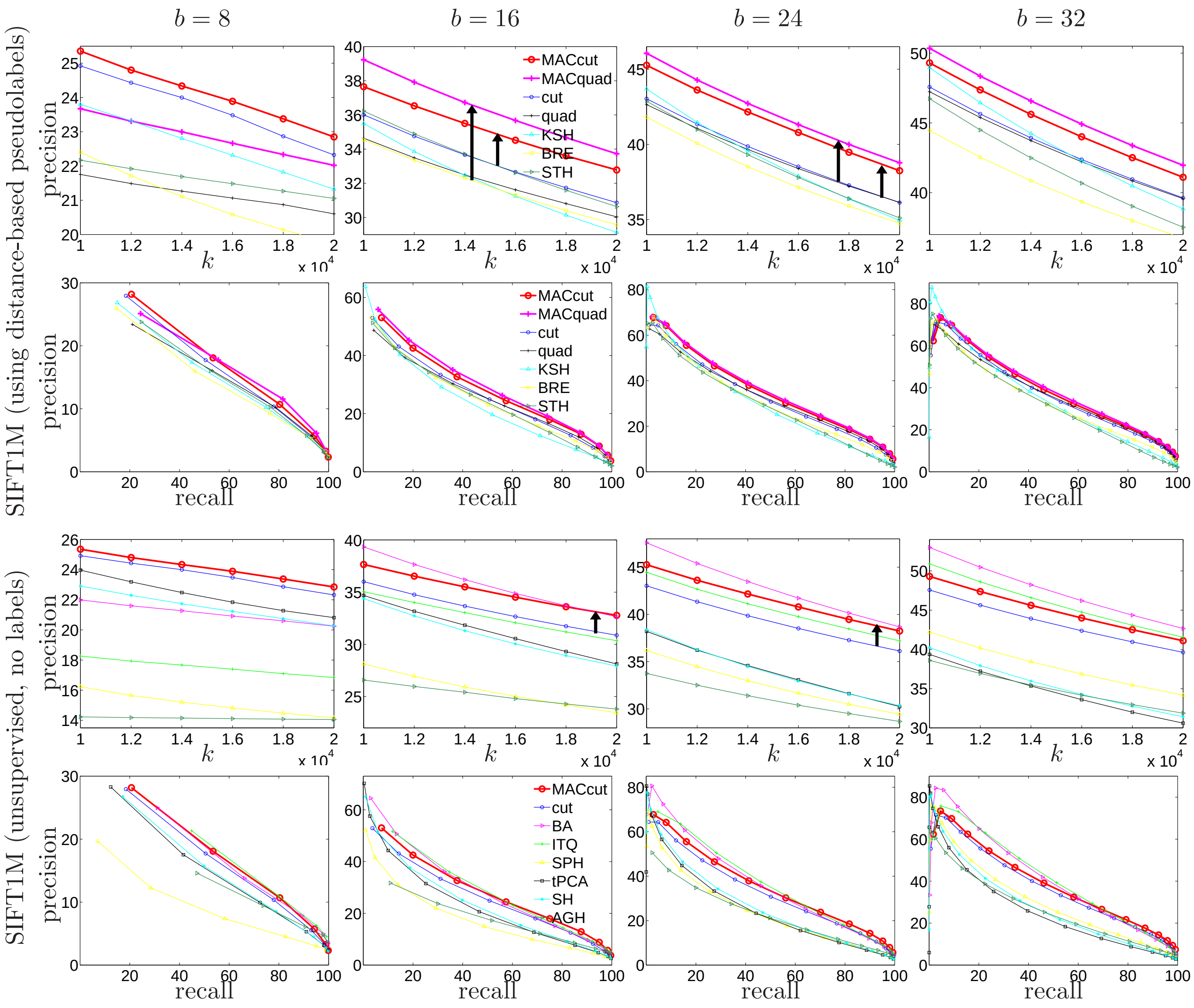

Figure 10: Comparison with binary hashing methods on SIFT1M using pseudolabels (top panel) and without labels (bottom panel). The rows in each panel show the precision (for a range of retrieved points k) and the precision/recall (at different Hamming distances), using $b = 8$ to 32 bits.

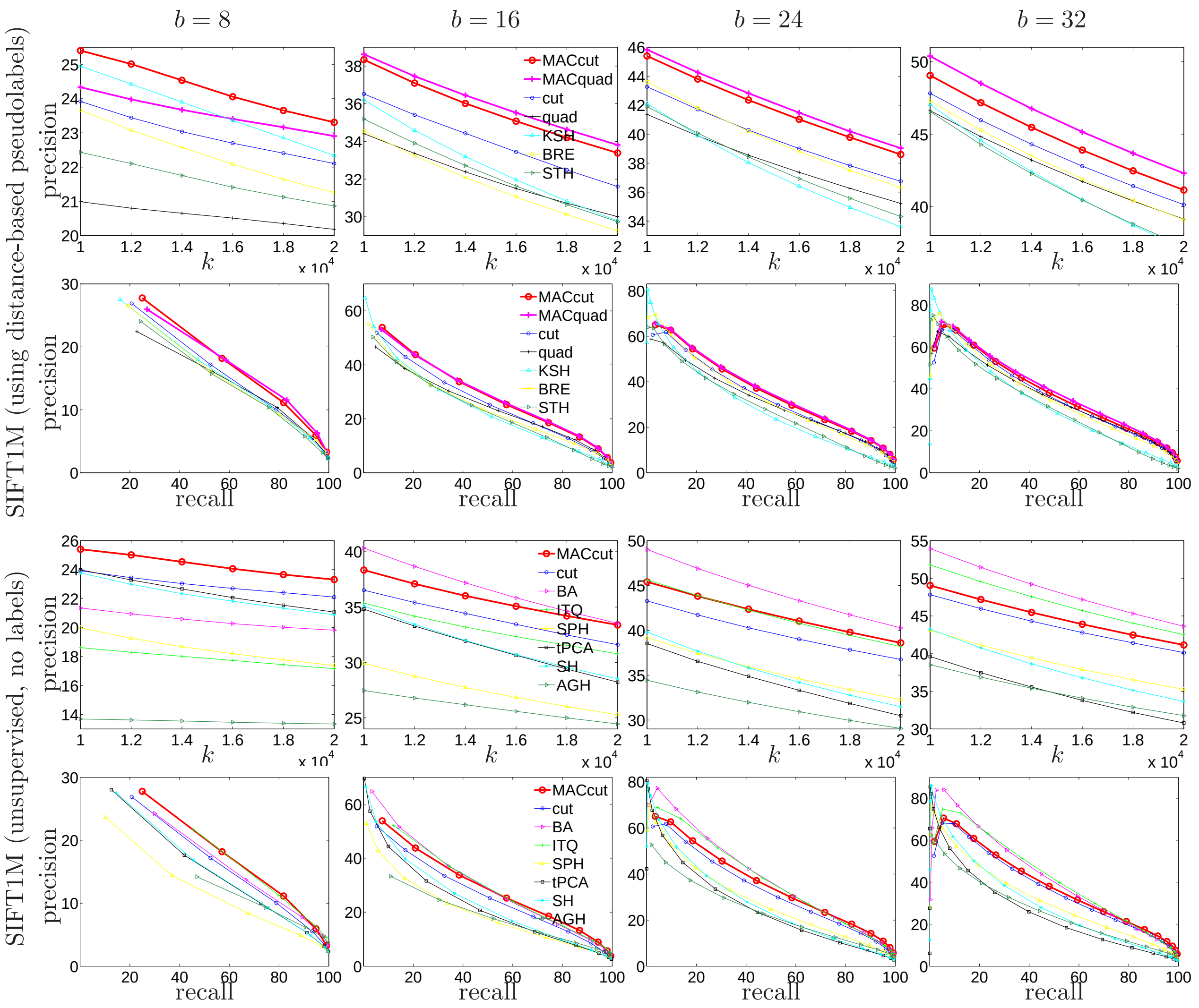

Figure 11: As in fig. 10 but using the cosine similarity instead of the Euclidean distance to find neighbors (i.e., all the points are centered and normalized before training and testing), on SIFT1M.

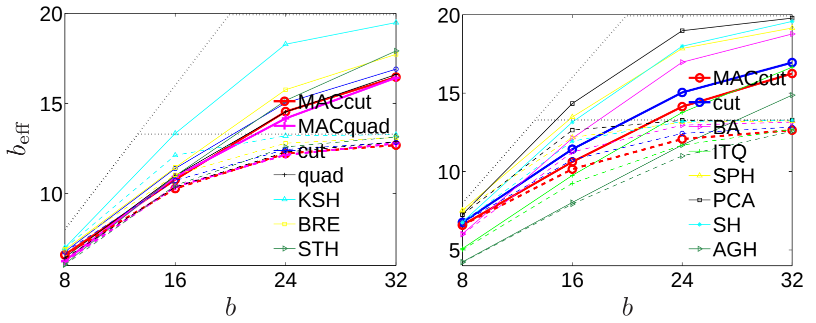

Figure 12: Code utilization in effective number of bits b_{eff} (entropy of code distribution) of different hashing algorithms, using $b = 8$ to 32 bits, for the SIFT1M dataset. The plots correspond to the codes obtained by the algorithms in figure 10, with solid lines for the training set and dashed lines for the test set. The two diagonal-horizontal black dotted lines give the upper bound (maximal code utilization) $\min(b, \log_2 N)$ on b_{eff} of any algorithm for the training and test sets (where N is the size of the training or test set).

Acknowledgments

Work supported by NSF award IIS–1423515. We thank Ming-Hsuan Yang, Yi-Hsuan Tsai and Mehdi Alizadeh (UC Merced) for useful discussions.

References

A. Andoni and P. Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Comm. ACM*, 51(1):117–122, Jan. 2008.

M. Belkin and P. Niyogi. Laplacian eigenmaps for dimensionality reduction and data representation. *Neural Computation*, 15(6):1373–1396, June 2003.

Y. Boykov and V. Kolmogorov. Computing geodesics and minimal surfaces via graph cuts. In *Proc. 9th Int. Conf. Computer Vision (ICCV'03)*, pages 26–33, Nice, France, Oct. 14–17 2003.

Y. Boykov and V. Kolmogorov. An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 26(9):1124–1137, Sept. 2004.

M. Á. Carreira-Perpiñán. The elastic embedding algorithm for dimensionality reduction. In J. Fürnkranz and T. Joachims, editors, *Proc. of the 27th Int. Conf. Machine Learning (ICML 2010)*, pages 167–174, Haifa, Israel, June 21–25 2010.

M. Á. Carreira-Perpiñán and R. Raziperchikolaei. Hashing with binary autoencoders. In *Proc. of the 2015 IEEE Computer Society Conf. Computer Vision and Pattern Recognition (CVPR'15)*, pages 557–566, Boston, MA, June 7–12 2015.

M. Á. Carreira-Perpiñán and M. Vladymyrov. A fast, universal algorithm to learn parametric nonlinear embeddings. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems (NIPS)*, volume 28, pages 253–261. MIT Press, Cambridge, MA, 2015.

M. Á. Carreira-Perpiñán and W. Wang. Distributed optimization of deeply nested systems. arXiv:1212.5921 [cs.LG], Dec. 24 2012.

M. Á. Carreira-Perpiñán and W. Wang. Distributed optimization of deeply nested systems. In S. Kaski and J. Corander, editors, *Proc. of the 17th Int. Conf. Artificial Intelligence and Statistics (AISTATS 2014)*, pages 10–19, Reykjavik, Iceland, Apr. 22–25 2014.

R.-E. Fan, K.-W. Chang, C.-J. Hsieh, X.-R. Wang, and C.-J. Lin. LIBLINEAR: A library for large linear classification. *J. Machine Learning Research*, 9:1871–1874, Aug. 2008.

T. Ge, K. He, and J. Sun. Graph cuts for supervised binary coding. In *Proc. 13th European Conf. Computer Vision (ECCV'14)*, pages 250–264, Zürich, Switzerland, Sept. 6–12 2014.

Y. Gong, S. Lazebnik, A. Gordo, and F. Perronnin. Iterative quantization: A Procrustean approach to learning binary codes for large-scale image retrieval. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 35(12):2916–2929, Dec. 2013.

K. Grauman and R. Fergus. Learning binary hash codes for large-scale image search. In R. Cipolla, S. Battiato, and G. Farinella, editors, *Machine Learning for Computer Vision*, pages 49–87. Springer-Verlag, 2013.

J.-P. Heo, Y. Lee, J. He, S.-F. Chang, and S.-E. Yoon. Spherical hashing. In *Proc. of the 2012 IEEE Computer Society Conf. Computer Vision and Pattern Recognition (CVPR'12)*, pages 2957–2964, Providence, RI, June 16–21 2012.

H. Jégou, M. Douze, and C. Schmid. Product quantization for nearest neighbor search. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 33(1):117–128, Jan. 2011.

R. Kohavi and G. H. John. The wrapper approach. In H. Liu and H. Motoda, editors, *Feature Extraction, Construction and Selection. A Data Mining Perspective.* Springer-Verlag, 1998.

V. Kolmogorov and R. Zabih. What energy functions can be minimized via graph cuts? *IEEE Trans. Pattern Analysis and Machine Intelligence*, 26(2):147–159, Feb. 2003.

A. Krizhevsky. Learning multiple layers of features from tiny images. Master's thesis, Dept. of Computer Science, University of Toronto, Apr. 8 2009.

B. Kulis and T. Darrell. Learning to hash with binary reconstructive embeddings. In Y. Bengio, D. Schuurmans, J. Lafferty, C. K. I. Williams, and A. Culotta, editors, *Advances in Neural Information Processing Systems (NIPS)*, volume 22, pages 1042–1050. MIT Press, Cambridge, MA, 2009.

B. Kulis and K. Grauman. Kernelized locality-sensitive hashing. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 34(6):1092–1104, June 2012.

G. Lin, C. Shen, D. Suter, and A. van den Hengel. A general two-step approach to learning-based hashing. In *Proc. 14th Int. Conf. Computer Vision (ICCV'13)*, pages 2552–2559, Sydney, Australia, Dec. 1–8 2013.

G. Lin, C. Shen, Q. Shi, A. van den Hengel, and D. Suter. Fast supervised hashing with decision trees for high-dimensional data. In *Proc. of the 2014 IEEE Computer Society Conf. Computer Vision and Pattern Recognition (CVPR'14)*, pages 1971–1978, Columbus, OH, June 23–28 2014.

W. Liu, J. Wang, S. Kumar, and S.-F. Chang. Hashing with graphs. In L. Getoor and T. Scheffer, editors, *Proc. of the 28th Int. Conf. Machine Learning (ICML 2011)*, pages 1–8, Bellevue, WA, June 28 – July 2 2011.

W. Liu, J. Wang, R. Ji, Y.-G. Jiang, and S.-F. Chang. Supervised hashing with kernels. In *Proc. of the 2012 IEEE Computer Society Conf. Computer Vision and Pattern Recognition (CVPR'12)*, pages 2074–2081, Providence, RI, June 16–21 2012.

G. Loosli, S. Canu, and L. Bottou. Training invariant support vector machines using selective sampling. In L. Bottou, O. Chapelle, D. DeCoste, and J. Weston, editors, *Large Scale Kernel Machines*, Neural Information Processing Series, pages 301–320. MIT Press, 2007.

D. G. Lowe. Distinctive image features from scale-invariant keypoints. *Int. J. Computer Vision*, 60(2): 91–110, Nov. 2004.

J. Nocedal and S. J. Wright. *Numerical Optimization.* Springer Series in Operations Research and Financial Engineering. Springer-Verlag, New York, second edition, 2006.

M. Norouzi and D. Fleet. Minimal loss hashing for compact binary codes. In L. Getoor and T. Scheffer, editors, *Proc. of the 28th Int. Conf. Machine Learning (ICML 2011)*, Bellevue, WA, June 28 – July 2 2011.

A. Oliva and A. Torralba. Modeling the shape of the scene: A holistic representation of the spatial envelope. *Int. J. Computer Vision*, 42(3):145–175, May 2001.

S. T. Roweis and L. K. Saul. Nonlinear dimensionality reduction by locally linear embedding. *Science*, 290 (5500):2323–2326, Dec. 22 2000.

G. Shakhnarovich, P. Indyk, and T. Darrell, editors. *Nearest-Neighbor Methods in Learning and Vision.* Neural Information Processing Series. MIT Press, Cambridge, MA, 2006.

C. Strecha, A. M. Bronstein, M. M. Bronstein, and P. Fua. LDAHash: Improved matching with smaller descriptors. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 34(1):66–78, Jan. 2012.

L. J. P. van der Maaten. Barnes-Hut-SNE. In *Int. Conf. Learning Representations (ICLR 2013)*, Scottsdale, AZ, May 2–4 2013.

L. J. P. van der Maaten and G. E. Hinton. Visualizing data using t-SNE. *J. Machine Learning Research*, 9: 2579–2605, Nov. 2008.

M. Vladymyrov and M. Á. Carreira-Perpiñán. Partial-Hessian strategies for fast learning of nonlinear embeddings. In J. Langford and J. Pineau, editors, *Proc. of the 29th Int. Conf. Machine Learning (ICML 2012)*, pages 345–352, Edinburgh, Scotland, June 26 – July 1 2012.

M. Vladymyrov and M. Á. Carreira-Perpiñán. Linear-time training of nonlinear low-dimensional embeddings. In S. Kaski and J. Corander, editors, *Proc. of the 17th Int. Conf. Artificial Intelligence and Statistics (AISTATS 2014)*, pages 968–977, Reykjavik, Iceland, Apr. 22–25 2014.

J. Wang, S. Kumar, and S.-F. Chang. Semi-supervised hashing for large scale search. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 34(12):2393–2406, Dec. 2012.

Y. Weiss, A. Torralba, and R. Fergus. Spectral hashing. In D. Koller, Y. Bengio, D. Schuurmans, L. Bottou, and A. Culotta, editors, *Advances in Neural Information Processing Systems (NIPS)*, volume 21, pages 1753–1760. MIT Press, Cambridge, MA, 2009.

Z. Yang, J. Peltonen, and S. Kaski. Scalable optimization for neighbor embedding for visualization. In S. Dasgupta and D. McAllester, editors, *Proc. of the 30th Int. Conf. Machine Learning (ICML 2013)*, pages 127–135, Atlanta, GA, June 16–21 2013.

S. X. Yu and J. Shi. Multiclass spectral clustering. In *Proc. 9th Int. Conf. Computer Vision (ICCV'03)*, pages 313–319, Nice, France, Oct. 14–17 2003.

D. Zhang, J. Wang, D. Cai, and J. Lu. Self-taught hashing for fast similarity search. In *Proc. of the 33rd ACM Conf. Research and Development in Information Retrieval (SIGIR 2010)*, pages 18–25, Geneva, Switzerland, July 19–23 2010.

C. Zhu, R. H. Byrd, P. Lu, and J. Nocedal. Algorithm 778: L-BFGS-B: FORTRAN subroutines for large-scale bound-constrained optimization. *ACM Trans. Mathematical Software*, 23(4):550–560, Dec. 1997.